Chain of Fools: An Exploration of Certificate Chain Validation Mishaps

Introduction

Typically, when software needs to leverage cryptography, developers use libraries or APIs that abstract many details away from them. They don't need to fully understand how TLS handshakes work to create a TLS socket, nor do they need to understand the cryptographic primitives used to encrypt SSH traffic when making SSH connections. However, some abstractions are leaky, and a better understanding is required to get things right. One example is validation of certificate chains, which is required when using APIs like Android SafetyNet or Android Protected Confirmation, validating remote attestations (like those provided by FIDO2 authenticators), or processing JSON Web Tokens that include signatures with associated X509 chains.

Applied cryptography can be hard even when using cryptographic libraries. For example, many cryptographic libraries make it difficult or non-obvious to properly validate certificate chains. Generally speaking, it's not because of defects in the implementation of primitives, but rather the difficulty of designing usable cryptographic APIs and providing clear, unambiguous documentation. The Internet is full of bad guidance regarding the implementation of common cryptographic workflows. Advice on validating certificate chains is no exception. Oftentimes, this advice instructs the developer to (unknowingly) add untrusted intermediates as trusted roots when building certificate chains, which breaks the chain of trust. This allows an attacker to provide an otherwise valid certificate chain that chains up to a "fake" root, which will cause certificate chain validation to succeed when it shouldn't.

An Observation: The Genesis of this Research

While we were working on a prototype that made use of the Android Protected Confirmation API, which includes a necessary step of validating an attestation certificate chain, we noticed that there wasn't an obvious way of safely validating such a certificate chain that includes untrusted intermediates with the pyOpenSSL Python module. We also recalled having a conversation with one of our colleagues, Adam Goodman, about this being a problem he ran into with the same module. In particular, when creating an `X509StoreContext` object with a depth greater than 1, (e.g., one which includes untrusted intermediates), the default behavior is to treat all certificates added to the `X509Store` object as trusted. As such, if a developer intends to build up a chain of trust by iteratively adding certificates to this store, starting with the root and `n` number of intermediates, and ultimately terminating in the leaf certificate, if any of the intermediates added to the `X509Store` object signed the leaf certificate, the validation will appear to have been successful. This is particularly troubling in the case when a developer is given `n` number of untrusted intermediates that could be controlled by an adversary, since they could artificially create a “fake” signing certificate that signs an untrustworthy leaf certificate. Even if the developer has made a policy decision ahead of time about which attestation roots are trusted, and they add those to the trust store, the chain of trust is broken and the certificate validation will succeed if an attacker manages to get a “fake” root in the list of untrusted intermediates which are added to the store.

```
from OpenSSL.crypto import load_certificate, load_privatekey
from OpenSSL.crypto import X509Store, X509StoreContext
from six import u, b, binary_type, PY3

root_cert_pem = b("""<snip>""")
intermediate_cert_pem = b("""<snip>""")
leaf_cert_pem = b("""<snip>""")

root_cert = load_certificate(FILETYPE_PEM, root_cert_pem)
intermediate_cert = load_certificate(FILETYPE_PEM,
intermediate_cert_pem)
leaf_cert = load_certificate(FILETYPE_PEM, leaf_cert_pem)
store = X509Store()
store.add_cert(root_cert)
store.add_cert(intermediate_cert)
store_ctx = X509StoreContext(store, leaf_cert)

# Will succeed if the intermediate signed the leaf, even if
# the root didn’t sign the intermediate.
print(store_ctx.verify_certificate())
```

Recognizing that this was an issue, and still needing a way to properly validate a certificate chain containing untrusted intermediates, Adam made a change to the pyOpenSSL module to add support for including a list of untrusted intermediates when constructing an `X509StoreContext` object, and submitted a pull request to the pyOpenSSL repository on GitHub. This was in June 2016, and as of this writing the PR has not been merged. It’s worth noting that this is not due to oversight or lack of caring by the maintainers of pyOpenSSL, but because the change is a sensitive one—and in a cryptographic library at that—they’ve been hesitant to merge the PR without first getting the right reviewers in front of it. This is a good reason.

Doing Things the “Right” Way With a Non-Obvious API

Although it’s not immediately obvious, (or at least it wasn’t to us and our colleague Adam), there is a way to correctly validate a certificate chain using the API mentioned above. The solution is simple, but the API doesn't really spell this out for us. Consider the following example from a test in the letsencrypt/boulder repository on GitHub:

```
def test_issuer():
    """
    Issue a certificate, fetch its chain, and verify the chain and
    certificate against test/test-root.pem. Note: This test only
handles chains
    of length exactly 1.
    """
    certr, authzs = auth_and_issue([random_domain()])
    cert = urllib2.urlopen(certr.uri).read()
    # In the future the chain URI will use HTTPS so include the root
certificate
    # for the WFE's PKI. Note: We use the requests library here so we
honor the
    # REQUESTS_CA_BUNDLE passed by test.sh.
    chain = requests.get(certr.cert_chain_uri).content
    parsed_chain =
OpenSSL.crypto.load_certificate(OpenSSL.crypto.FILETYPE_ASN1, chain)
    parsed_cert =
OpenSSL.crypto.load_certificate(OpenSSL.crypto.FILETYPE_ASN1, cert)
    parsed_root =
OpenSSL.crypto.load_certificate(OpenSSL.crypto.FILETYPE_PEM,
        open("test/test-root.pem").read())

    store = OpenSSL.crypto.X509Store()
    store.add_cert(parsed_root)

    # Check the chain certificate before adding it to the store.
    store_ctx = OpenSSL.crypto.X509StoreContext(store, parsed_chain)
    store_ctx.verify_certificate()
    store.add_cert(parsed_chain)

    # Now check the end-entity certificate.
    store_ctx = OpenSSL.crypto.X509StoreContext(store, parsed_cert)
    store_ctx.verify_certificate()
```

In the above example, the intermediate certificate is validated with the root before being added to the `X509Store` object. If the intermediate validation passes, only then will it be added to the trust store and used to validate the end-entity certificate. Even though this works, as it does correctly validate the certificate chain, it only supports chains of length 1. Although it would be possible to update this function to support chains of greater depth, the code could get messy quickly, and the opportunity for mistakes to arise would only increase. One of the biggest problems here is that the pyOpenSSL API simply refers to the operation of adding a cert to the `X509Store` object as `add_cert`. This is problematic because that API makes sense if the developer makes the correct assumption about how it works, but it can be hugely problematic if they make the wrong assumption, as we did. For example, if the method were renamed to `add_trusted_cert` or similar, this would be incorrect if the developer uses the API "correctly." Perhaps one way of combating this would be to only allow the developer to add one cert at a time to the `X509Store`, but this has other usability limitations, and sometimes could be undesirable, (for example in unit tests).

Bad Advice on the Internet

Google's SafetyNet documentation gives developers the following steps they must follow to verify the origin of a signed SafetyNet attestation:

> To verify the origin of the JWS message, complete the following steps:

> 1. Extract the SSL certificate chain from the JWS message.
> **2. Validate the SSL certificate chain and use SSL hostname matching to verify that the leaf certificate was issued to the hostname attest.android.com.**
> 3. Use the certificate to verify the signature of the JWS message.
> 4. Check the data of the JWS message to make sure it matches the data within your original request. In particular, make sure that the timestamp has been validated and that the nonce, package name, and hashes of the app's signing certificate(s) match the expected values.

The example code in the SafetyNet server example Java project uses `javax.net.ssl.TrustManager` under the hood to validate the certificate chain by means of using Google's `JsonWebSignature` library. The default trust manager chains up to the system roots of trust, and the example code validates that a path can be built between the leaf certificate provided by SafetyNet up to a system root of trust, likely through intermediate certificates that are also provided in the SafetyNet attestation response. The example code does things right, but there are many steps performed during the validation process that are not explicitly stated by the documentation.

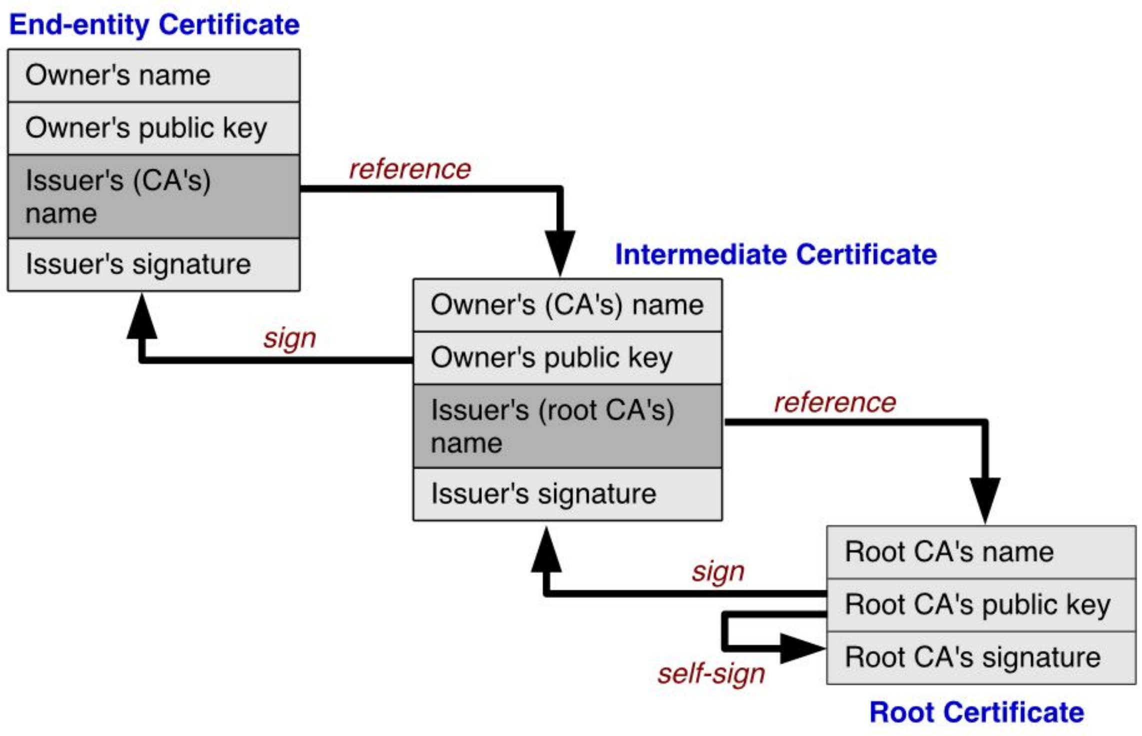


Image Source: Wikipedia

Looking at results for “validate SSL certificate chain” on sites like StackOverflow and GitHub, we see a worrying pattern. Let’s look at several of these examples.

The first example we’ll look at is one of the first results for “Golang verify certificate chain” on Google, which you can find here. In particular, the following lines are used to import the intermediate certificates, which are then used as trusted roots:

```
rootPEM, err := ioutil.ReadFile(os.Args[3]) // cert-chain PEM
if err != nil {
	log.Fatal(err)
}

roots := x509.NewCertPool()
ok := roots.AppendCertsFromPEM([]byte(rootPEM))
if !ok {
	panic("failed to parse root certificate")
}
```

This method of validating a certificate chain implicitly trusts all the intermediates in the chain. If there exists a self-signed certificate in the chain, it will be treated as a trusted root, regardless of system trust settings. As a result, an intermediary could modify payload contents, re-sign the payload with a new key, and then attach the new leaf certificate and issuing CA to the request.

Searching for a method to validate certificate chains using Python might lead you to this StackOverflow question. The first response, published in 2015, advises the question's author to use pyOpenSSL's `X509StoreContext` to validate the chain, which falls victim to the issues described earlier. This is in part due to incomplete documentation and bad advice on the internet, and in part due to unit tests for pyOpenSSL that perform the likely-undesirable validation of a certificate with intermediates as trusted roots.

This StackOverflow question submission asks how to verify a certificate chain using the `openssl verify` command. In this case, there is a correct response (instructing the poster to use the `-untrusted` option), but there is also another high-rated answer that suggests using `cat` to combine the roots and intermediates into a single list of certificates, all of which will be considered trusted roots. A question attached to this answer asks "Will this actually verify the intermediate cert against the root cert?" to which someone replies "It does. I just re-ran the commands with a chain that I know is correct."

Joel Sandin brought this same issue up on the cryptography-dev mailing list in August of 2016. He says: "It's clear from what I've found online that developers are confused and may have introduced vulnerabilities into their code." It is fair to assume this is the case—it seems relatively unlikely that users who follow the misleading advice would check back in once updated answers were posted calling out the dangers of the incorrect methods of certificate chain validation. The failure mode is insidious: there are no false negatives introduced by accidentally treating intermediate certificates as trusted roots, so unless validation methods are explicitly tested with input that includes a self-signed CA in the list of intermediates, this issue can go undetected.

This raises an important point. Software developers typically have deadlines. Oftentimes, they're incentivized to get their code working, not necessarily to get it right. When you hit a wall when trying to solve a tough problem, and you finally get it working, how often do you ask yourself if you solved the problem the right way? This reminds me of the following sentiment about losing your keys:

"It's frustrating to feel like your lost keys are always in the last place you look, but the alternative is to keep looking for them after you've found them."

- u/droidpat on r/Showerthoughts, Reddit

That's the thing. When doing anything with cryptography in software, you've got to keep looking for your keys even after you've found them.

Misuse-resistant APIs

Security practitioners have historically tended to blame the user: every single security issue, after all, is due to some sort of “human error.” It’s a tidy excuse that allows the conversation to end there, but a new, more holistic model of security is growing. This new model considers how easy it is for a developer to misuse a primitive. This is particularly important in the field of cryptography, where seemingly small mistakes can have massive consequences (ex: nonce reuse).

Misuse resistance is relevant at all levels, from design of primitives up to the high-level APIs. AES-GCM-SIV, for example, is designed to minimize the impact of nonce misuse. EdDSA signatures are deterministic, protecting against the kinds of mistakes that caused the leak of the PS3 content signing key via a reuse of `k` values in the ECDSA scheme.

Miscreant is a library developed by Tony Arcieri with the explicit goal of providing misuse-resistant APIs for common cryptographic tasks. Miscreant consists of libraries in five different languages implementing misuse-resistant primitives. Other libraries (such as `libsodium`) may be built on primitives that are not misuse resistant, but expose APIs such that misusing the primitives is hard or impossible.

Many older cryptographic primitives are still used purely due to inertia. Trail of Bits points this out well in their *Seriously, stop using RSA* post: there are myriad ways to misuse RSA, many of which are due to subtle implementation concerns that developers would not know about without stronger cryptography knowledge. But rather than lamenting that developers are not sufficiently knowledgeable about padding oracles, the path forward involves building tooling that does not require domain-specific knowledge to use.

Misuse Resistance and Legacy Cryptography

X509 parsing is a minefield, and the tooling built around it (much of it bindings to OpenSSL) has far too many knobs for the average developer to build safe applications. If you are expecting developers to implement certificate chain validation, providing explicit instructions is a minimum requirement, and providing tooling to do so is preferable. This is a rule of thumb that extends to any situation in which you are asking developers to perform some cryptographic step: we must ask “what knowledge is needed to do this correctly?”, “what is the impact of misuse or improper implementation?”, and “how can we reduce or eliminate the risk caused by lack of developer cryptographic knowledge?”.

Ideally, there should be no requirement that developers manually implement these steps: a service provider should maintain client libraries that implement all the necessary cryptographic checks without exposing intricacies to the developer when it’s not necessary. Google does a version of this by providing an online validation API for SafetyNet tokens, but it exists only for development purposes, and is heavily rate limited.

Quantifying the Use of SafetyNet

Knowing that certificate chain validation is hard to get right, and that it's a necessary step of validating SafetyNet attestations, we set out to understand how widely used this API is among among popular Android applications, and among applications in different categories. The reasons for this are twofold. For one, knowing how widely used this API is useful in its own right, and two, because of the necessary certificate chain validation step, knowing how popular this API is will give us an idea of how many implementations could potentially be vulnerable to incorrect certificate chain validation logic. At a high level, answering these questions required the following steps: 1) acquiring a list of popular Android applications by number of installs and a ranking of applications, 2) downloading the Android Packages (APKs) associated with these applications en masse, and 3) analyzing the corpus of APKs to determine the use of the Safetynet Attestation API.

In the following sections, we describe the methods we used to accomplish the three aforementioned steps.

Acquiring a List of Top Android Apps

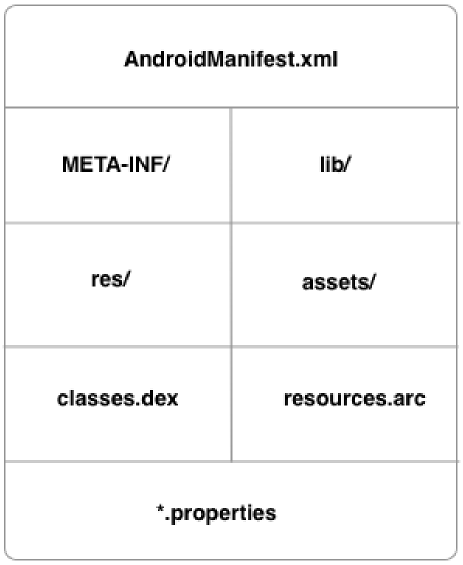
In order to find out which of the top Android apps are using the SafetyNet Attestation API, we first needed to acquire a source that outlined what those top apps were. In our search for a reliable source, we found that there were some smaller lists that detailed dozens of apps, and there were paid app store analytics services that also provided these sorts of lists. However, we wanted our work to be reproducible by other researchers; so, we opted to use Android Rank, which has been collecting application metrics since 2011 and has been used by others for similarly large analyses. Using data from their site, we were able to obtain a list of the most installed Android applications and assemble a list of the top Android applications for 32 general application categories (e.g., Communication, Finance, Social, etc.) and 17 categories of games (e.g., Action, Puzzles, Sports, etc.). In total, this resulted in a list of over 24,296 apps.

Building a Corpus of Applications

After compiling a list of Android applications, the next thing that we had to do was to acquire as many of the 24,000 APKs as possible. Currently, Google does not allow for automated downloading of APKs from the Google Play Store; moreover, you are restricted to downloading apps for your particular region. As a result, we decided to build our corpus of applications from APK downloading services such as APKPure and apkmonk. These sites allow users to download APKs for both current and older versions of apps, apps from different regions, etc., so that they can be sideloaded, or in our case, analyzed. Because downloading such a large number of applications by hand was intractable, we opted to automate this process. We did so by building off the work done by the Open GApps Project, which downloads Google Apps packages from APK downloading services like the two we mentioned above. Using the scripts that we created, we were able to download 98% (23768/24296) of these applications. We were unable to download all of these applications because of two reasons: 1) we did not limit our list to free apps and 2) AndroidRank's list of the most popular applications includes apps that are no longer available in general, or on the two sites that we used.

Analyzing APKs

In order to determine the use of SafetyNet by a particular application, we performed a series of checks of the applications that is progressively more intensive. Before getting started, it's helpful to describe what an APK is and what it looks like under-the-hood. APKs can be thought of as zip files that contain the code for the application, the resources, assets and other files as shown in the diagram below.



Modified version of diagram by Ryantzj

At this top level and after the APK is unzipped, is where our first set of checks for the use of the SafetyNet happen. We start by checking for the existence of a SafetyNet properties file. Properties files are often used to store configuration information for applications. As it relates to this work, there will sometimes be a file named `play-services-safetynet.properties`. This file will contain information relating to the version of SafetyNet that is being used by the application. It is important to note that the existence of this file only means that one of the four SafetyNet APIs is being used, but it doesn't tell us if the one being used is the Attestation API. In addition to the SafetyNet Attestation API, there are SafetyNet APIs for Safe Browsing, reCAPTCHA, and for verifying apps. Nevertheless, in our process, we store this information so that we can catalog which versions of SafetyNet are being used.

After checking for the existence of a SafetyNet properties file, we analyze the contents of the Android Manifest. The Android Manifest is an XML file that contains information such as the app's package name, the activities and services used by the app, the permissions that app needs to operate, etc. In the Android Manifest, we are specifically looking for the key `com.google.android.safetynet.ATTEST_API_KEY.`

Last, if we are unable to ascertain the use of the SafetyNet Attestation API, we analyze the `classes.dex` file, searching specifically for the presence of "`AttestationResponse`" or an attestation API Key.

Initial Results

Prior to our work, it was found that less than one percent of Android applications were using one of the safetynet APIs among a sample of 3000 applications. This study took place back in 2017, which was just two years after SafetyNet was first introduced in 2015. Our study showed that the use of SafetyNet has increased since then to approximately 5.26% of the approximately 24 thousand apps that we were able to successfully analyze. The category that showed the largest use of SafetyNet was Finance at 18.52% while the lowest group was Tools (e.g., flashlight apps) at 1.62%. Among the most installed applications according to Android Ranks, the use of SafetyNet proved to be higher than average at 10.82%

Application Category	Percent using SafetyNet
Finance	18.52
Comics	12.63
Dating	11.00
Shopping	9.85
Gaming	5.23

Top 5 categories by SafetyNet usage

Application Category	Percent using SafetyNet
Books and Reference	2.61
Tools	2.42
House and Home	2.42
Personalization	2.22

Weather	2.09

Bottom 5 categories by SafetyNet usage

In addition, we found that only 23% of the apps that we found to be using SafetyNet were using the latest version of the SafetyNet API, `17.0.0`. The vast majority (62%) were using the previous version of the API: `16.0.0`. Since the latest version was released on June 17th, we expect the percentage of applications using version `17.0.0` to increase.

Using More Advanced Approaches

Initially, when we began this research, we focused on trying to detect the use of the SafetyNet APIs by file existence and pattern matching as mentioned above. This approach has its drawbacks, with one of the largest being that the method calls that we were looking for could be obfuscated, thus thwarting any direct or fuzzy string matching. In addition, by relying on the presence of the properties file to determine which versions of safetynet are being used, our figures would be biased in respect to the applications that use properties files as a configuration mechanism. Because of these limitations, we turned to other library detection mechanisms.

Specifically, we decided to use LibScout. LibScout works by first extracting profiles from original versions of a library. In our case, this would be the `.jars` for the different version of `com.google.android.gms.play-services-safetynet`. Using the profiles, LibScout statically detects the use of libraries in Android apps by building application profiles and then applying a pattern matching algorithm to check for the presence of a given library. If a given library is deemed to be present, LibScout outputs a similarity score between 0 and 1 where 1 means that a given library version was matched exactly.

Using LibScout, we analyzed a sample of 7,832 applications to see how the results compared to purely searching for strings within the APK file. We found that a higher percentage of applications were using the SafetyNet library at 7.2%. Similar to our analysis of the versions of the SafetyNet library that relied on the properties file, we found that most (86%) of the applications that we analyzed were not using the latest version. Last, we found that the most popular app category that was using SafetyNet was Gaming at 11.2%.

Limitations

It is important to note that while our corpus of apps is larger than the study run in 2017, it is small compared to the entire population. Moreover, the sample that we collected contains more apps that can be categorized as games than any other category. Therefore, these results can’t be generalized to the entire Android ecosystem.

Research Methodology

When setting out to perform this research, we decided to approach the problem from a number of different angles simultaneously. In addition to determining how widely used the SafetyNet Attestation API is in certain verticals, we also wanted to explore what it would take to forge SafetyNet attestations. While some of us were looking into the possibility of modifying the SafetyNet attestation from the stance of a network MITM, our colleague Mikhail Davidov was looking into doing so through in-process hooking with the Frida instrumentation framework. Ultimately, we focused primarily on forging attestations through a mitmproxy Addon, but did confirm that this should also be possible through in-process hooking with Frida. In the spirit of sharing as much of our research process as possible, so as to enable further investigation by other researchers, we're including the following snippet below.

```
Java.perform(function() {
  var attResp =
Java.use('com.google.android.gms.safetynet.SafetyNetApi$AttestationRes
ponse');
attResp.getJwsResult.implementation = function() {
     var jwsToken = this.getJwsResult()
     console.log(jwsToken);
     // Change the token here.
     return jwsToken;
  }
},0);
```

Proof of Concept

To demonstrate this issue in action, we wrote several utilities:

- An Android app that requests a SafetyNet attestation
- A Flask web app that receives and incorrectly validates attestations
- A tool to modify and re-sign the attestation, including a mitmproxy Addon

You can access these three utilities on GitHub. Our Flask application makes use of the `X509Context` pattern that led to the investigation into these issues in the first place. `jwsmodify` contains a method `modify_jws_and_forge_signature`, which takes a JWS payload in raw bytes as input and performs a transformation. The transform function is used to modify the payload (to, for example, set passing SafetyNet values by setting `ctsProfileMatch` and `basicIntegrity` to `true`). Finally, the payload is re-signed by a newly-generated self-signed CA that is encoded back into the JWS. This allows us to forge SafetyNet assertions.

We include a mitmproxy Addon with the JWS modification tools. By default, the Addon looks for SafetyNet attestations and sets them to passing. Our Flask application will validate both unmodified attestations and attestations modified by our tools.

While our proof of concept tooling focuses on SafetyNet attestations in particular, the same tools should be applicable to the modification of any JWS payload that uses the `x5c` parameter to bundle certificate chains with the signature.

How We Can Make Things Better

There are a number of steps that developers can take to prevent improper certificate chain validation or other cryptography pitfalls.

- Use high-level libraries and abstractions wherever possible. Ideally, these libraries will shield you from having to deal with anything cryptographic. For example, the NSURLSession API for iOS and macOS allows developers to download data from or upload data to HTTPS URLs, but doesn't require knowledge of the lower level cryptographic primitives that it builds upon.
- If you need to interact directly with cryptographic libraries or APIs, choose misuse resistant ones where possible, such as Miscreant and `libsodium`.
- When doing anything with cryptography, don't assume because something works that it's correct. In other words, keep looking for your keys even after you've found them. ;)
- Ask for help! This can come in the form of requesting additional documentation or clarification from vendors, or by bringing up questions in online communities.
- Be skeptical. The stakes can be pretty high when getting things wrong with cryptography, and the failure modes are often subtle.

Also, vendors should take great care when requiring low-level cryptographic steps of any kind from developers integrating with their APIs. Ideally, they should provide client libraries that abstract as many of these details away from the user, and failing that should provide detailed documentation.

We can also work to prevent these kinds of mishaps as a security community, by educating developers and providing help when we see it's needed, (hat tip to Scott Arciszewski of Paragon Initiative for his contributions to the Stack Overflow community, among other things). Making suggestions without assigning blame not only helps individuals, but it also improves the perception of security practitioners in the public eye, hopefully making people less shy about asking questions in the future.

Next Steps for Research

In addition to testing incorrect certificate chain validation implementations with the Android SafetyNet API, this research should also apply anywhere where untrusted intermediate certificates are provided through an untrusted channel and used to build a chain of trust, then validated against a root of trust. Other examples that come to mind are when verifying a signed message from the Android Protected Confirmation API, or when validating a WebAuthn attestation statement. Additionally, it may be possible to use the steps outlined in this research to test incorrect certificate chain validation implementations at scale by modifying JWS, (or other payloads containing untrusted intermediate certificates), en masse. Finally, scanning public source code repositories for faulty certificate chain validation logic could surface interesting results.

Conclusion

It's no secret that it's easy to get things wrong when leveraging cryptography in software, but there are concrete steps that can be taken by developers, vendors, cryptographic library authors, and security practitioners to prevent mistakes from happening. In this research, we've demonstrated how things like certificate chain validation can go awry, and what that means in practice when developers are tasked with using or writing cryptographic code with insufficient information or expertise. Forging Android SafetyNet attestations by taking advantage of incorrect certificate chain validation implementations is just one example of how things can go wrong when incorrect assumptions are made in cryptographic systems. Like many things in security, humans are core to the experience. By working together as a community and not assigning blame when things go wrong, we can work toward a more secure future.